Análisis Exploratorio y Segmentación de Datos – *Sample 101 Clean Dataset*

Contexto general

En el análisis de datos, uno de los pasos fundamentales antes de aplicar cualquier modelo predictivo o de agrupamiento es realizar un **análisis exploratorio de los datos (EDA)**.
Este proceso permite identificar patrones, relaciones entre variables, valores atípicos y posibles errores en la información.

El dataset utilizado en este proyecto, `sample_101_clean_dataset.csv`, contiene datos previamente depurados que servirán como base para la exploración y posterior modelado.
Dependiendo del contexto, este conjunto puede representar información de clientes, productos, transacciones u observaciones experimentales.

Objetivos del análisis

1. **Comprender la estructura del dataset:** número de filas, columnas, tipos de datos y variables principales.
2. **Explorar relaciones entre variables:** identificar correlaciones, tendencias y distribuciones.
3. **Detectar valores atípicos o faltantes:** y decidir cómo tratarlos de forma apropiada.
4. **Preparar los datos para modelado:** mediante limpieza, transformación y escalado de variables si es necesario.
5. **Realizar una segmentación preliminar o clustering** (si aplica), con el fin de agrupar elementos similares según su comportamiento o características.

Metodología general

1. **Carga del dataset** desde archivo `.csv` y verificación de su integridad.
2. **Inspección general:** revisión de tipos de datos, conteo de nulos y estadísticos descriptivos.
3. **Visualización inicial:** histogramas, diagramas de dispersión y matrices de correlación para detectar patrones relevantes.
4. **Normalización / Estandarización de datos:** preparación para algoritmos de Machine Learning.
5. **Aplicación de clustering o análisis estadístico:** según los objetivos del estudio.

Resultado esperado

Tras completar este notebook, se obtendrá una comprensión clara de la estructura del dataset y una **base sólida para realizar análisis avanzados** (como clasificación, regresión o segmentación).
Este trabajo también servirá para garantizar que los datos sean **coherentes, completos y representativos** antes de pasar a etapas posteriores de modelado.

> **Nota:** El análisis exploratorio no busca generar predicciones, sino **comprender los datos**.
> Esta comprensión es clave para construir modelos precisos, interpretables y útiles en contextos de negocio o investigación.

In [1]:
```
import pandas as pd

# Cargar archivo CSV
df = pd.read_csv("sample_101_clean_dataset.csv")

# Seleccionar columnas relevantes para el modelo
df_model = df[["Age", "Income", "Approved"]].dropna()

# Separar X e y
X = df_model[["Age", "Income"]].values.astype(float)
y = df_model["Approved"].values.astype(int)

# Verificar dimensiones
print(X.shape, y.shape)
print(df_model.head())
```

```
(100, 2) (100,)
     Age  Income  Approved
0  30.83       0         1
1  58.67     560         1
2  24.50     824         1
3  27.83       3         1
4  20.17       0         1
```

División de datos y escalado (preprocesamiento)

Antes de entrenar el modelo, es fundamental **dividir los datos** en conjuntos de entrenamiento y prueba, garantizando que el modelo aprenda de una parte de los datos y se evalúe con ejemplos no vistos.
Además, las variables se **escalan mediante Min-Max Scaling** para normalizarlas en un rango de [0, 1]. Esto mejora la estabilidad del algoritmo Perceptrón, evitando que las diferencias de escala afecten los resultados.

Pasos realizados:

- Separación en `train` (70%) y `test` (30%), conservando la proporción de clases (`stratify`).
- Aplicación del **escalador MinMaxScaler** de `scikit-learn`.

- Visualización de las primeras filas escaladas.

In [2]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# División en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)

# Escalado Min-Max
scaler = MinMaxScaler()
X_train_sc = scaler.fit_transform(X_train)
X_test_sc = scaler.transform(X_test)

# Mostrar ejemplos
print(X_train_sc[:5])
```

```
[[0.07796452 0.        ]
 [0.17507003 0.01985703]
 [0.52334267 0.        ]
 [0.15756303 0.03706645]
 [0.28011204 0.00231665]]
```

Entrenamiento del Perceptrón simple

El **Perceptrón** es uno de los modelos más básicos de aprendizaje automático supervisado.
Su objetivo es encontrar una frontera lineal que separe correctamente las clases.

En esta celda:

- Se define la función `entrenar_perceptron` con:
  - Tasa de aprendizaje (`lr`).
  - Número máximo de épocas (`epocas`).
- Se actualizan los pesos `W` y el sesgo `b` en cada iteración según el error de predicción.
- El entrenamiento se detiene si no hay errores en una época completa.

Finalmente, se muestran:

- Los **pesos finales** y el **bias (b)**.
- El número de **errores cometidos por época**, útil para observar la convergencia del modelo.

In [3]:
```
import numpy as np

def entrenar_perceptron(Xm, ym, lr=0.05, epocas=300, seed=42):
    rng = np.random.default_rng(seed)
    W = rng.uniform(-1, 1, size=Xm.shape[1])
    b = rng.uniform(-1, 1)
    errores_epoca = []

    for _ in range(epocas):
        errores = 0
```

```
        for xi, yi in zip(Xm, ym):
            z = np.dot(W, xi) + b
            y_hat = 1 if z > 0 else 0
            error = yi - y_hat
            if error != 0:
                W += lr * error * xi
                b += lr * error
                errores += 1
        errores_epoca.append(errores)
        if errores == 0:
            break
    return W, b, errores_epoca

# Entrenamiento
W, b, errores = entrenar_perceptron(X_train_sc, y_train)

# Mostrar pesos y errores por época
print("Pesos finales:", W)
print("Bias final:", b)
print("Errores por época:", errores)
```

```
Pesos finales: [0.05565019 0.64869612]
Bias final: -0.0328041601772352
Errores por época: [23, 28, 28, 24, 26, 26, 24, 26, 24, 26, 26, 24, 24, 22, 24, 2
2, 22, 24, 22, 24, 22, 24, 22, 22, 24, 22, 24, 22, 24, 22, 22, 22, 22, 22, 22, 2
2, 22, 22, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 2
2, 22, 22, 24, 22, 24, 22, 22, 22, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 2
2, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 22, 22, 2
2, 22, 22, 24, 22, 22, 22, 22, 22, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 2
2, 22, 22, 24, 22, 24, 22, 22, 22, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 2
2, 22, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 22, 24, 22, 22, 22, 24, 22, 24, 2
2, 22, 22, 22, 22, 24, 22, 24, 22, 24, 22, 24, 22, 22, 24, 22, 24, 22, 24, 22, 2
4, 22, 24, 22, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 2
2, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 2
4, 22, 24, 22, 22, 24, 22, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 2
2, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 2
4, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 2
4, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 22, 24, 24, 2
2, 24, 24, 22, 24]
```

Evaluación de combinaciones de atributos

En esta sección se prueba el rendimiento del Perceptrón utilizando diferentes pares de variables numéricas del dataset (por ejemplo, *Income*, *Debt*, *CreditScore*, etc.).

Objetivo:

- Determinar qué combinación de variables ofrece **mejor separación lineal entre clases**.

Pasos principales:

1. Se generan todas las combinaciones posibles de pares de columnas.
2. Para cada par:
   - Se entrena un modelo con `train_perc`.
   - Se mide su precisión (`accuracy`).

3. Se guarda el mejor resultado y los pesos correspondientes.
4. Se muestran los **5 pares con mejor rendimiento** y el **mejor par global**.

Este proceso permite identificar qué variables son más informativas para el modelo lineal.

In [4]:
```
import numpy as np
import pandas as pd
from itertools import combinations
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score

# columnas numéricas candidatas (ajusta si quieres)
num_cols = ["Age","Income","Debt","YearsEmployed","CreditScore"]
target = "Approved"

df_ = df[num_cols+[target]].dropna().copy()
X_all = df_[num_cols].values.astype(float)
y_all = df_[target].astype(int).values

def train_perc(Xtr, ytr, lr=0.05, ep=400, seed=42):
    rng = np.random.default_rng(seed)
    W = rng.uniform(-1,1, size=Xtr.shape[1]); b = rng.uniform(-1,1)
    for _ in range(ep):
        err=0
        for xi,yi in zip(Xtr,ytr):
            yhat = 1 if (np.dot(W,xi)+b)>0 else 0
            e = yi - yhat
            if e!=0:
                W += lr*e*xi; b += lr*e; err+=1
        if err==0: break
    return W,b

def eval_pair(c1,c2):
    X = df_[[c1,c2]].values.astype(float)
    y = df_[target].astype(int).values
    Xtr,Xte,ytr,yte = train_test_split(X,y,test_size=0.3,stratify=y,random_state
    sc = MinMaxScaler(); Xtr = sc.fit_transform(Xtr); Xte = sc.transform(Xte)
    W,b = train_perc(Xtr,ytr,lr=0.05,ep=800,seed=42)
    yhat = (Xte@W + b > 0).astype(int)
    acc = accuracy_score(yte,yhat)
    return acc, (W,b,sc)

best = (-1,None,None)
scores = []
for c1,c2 in combinations(num_cols,2):
    acc,(Wb,bb,scb) = eval_pair(c1,c2)
    scores.append((acc,c1,c2))
    if acc>best[0]:
        best = (acc,(c1,c2),(Wb,bb,scb))

scores_sorted = sorted(scores, reverse=True)
print("TOP 5 pares:", scores_sorted[:5])
print("MEJOR PAR:", best[1], "ACC=", round(best[0],3))
best_pair = best[1]; (W_best,b_best,sc_best) = best[2]
```

```
TOP 5 pares: [(0.8333333333333334, 'Income', 'CreditScore'), (0.7, 'Age', 'Credit
Score'), (0.6666666666666666, 'YearsEmployed', 'CreditScore'), (0.666666666666666
6, 'Debt', 'CreditScore'), (0.6333333333333333, 'Income', 'YearsEmployed')]
MEJOR PAR: ('Income', 'CreditScore') ACC= 0.833
```

Balanceo de clases y limpieza de valores extremos

Dado que el dataset puede tener **clases desbalanceadas** (más ejemplos de una clase que de otra), se aplica una estrategia de balanceo mediante muestreo:

- Se iguala el número de ejemplos de cada clase.
- Se mezclan las observaciones aleatoriamente para evitar sesgos.

Además:

- Se recortan los valores extremos (percentiles 1% y 99%) para evitar que outliers distorsionen la frontera de decisión.
- Se reescala nuevamente con `MinMaxScaler` para mantener consistencia.

Este paso garantiza que el modelo aprenda con datos equilibrados y sin valores extremos que afecten su desempeño.

In [5]:
```
# aplica al mejor par encontrado
c1,c2 = best_pair
df_bal = df_[[c1,c2,target]].dropna().copy()
maj = df_bal[df_bal[target]==df_bal[target].value_counts().idxmax()]
minr = df_bal[df_bal[target]!=df_bal[target].value_counts().idxmax()]

rep = int(len(maj)/len(minr)) - 1
df_balanced = pd.concat([maj, minr] + [minr]*max(rep,0), ignore_index=True).samp

Xb = df_balanced[[c1,c2]].values.astype(float)
yb = df_balanced[target].astype(int).values

Xtr,Xte,ytr,yte = train_test_split(Xb,yb,test_size=0.3,stratify=yb,random_state=
sc = MinMaxScaler(); Xtr = sc.fit_transform(Xtr); Xte = sc.transform(Xte)
```

In [6]:
```
lo1, hi1 = np.percentile(Xtr[:,0],[1,99])
lo2, hi2 = np.percentile(Xtr[:,1],[1,99])
Xtr[:,0] = np.clip(Xtr[:,0], lo1, hi1)
Xtr[:,1] = np.clip(Xtr[:,1], lo2, hi2)
Xte[:,0] = np.clip(Xte[:,0], lo1, hi1)
Xte[:,1] = np.clip(Xte[:,1], lo2, hi2)
```

Perceptrón con memoria – Pocket Algorithm

El algoritmo **Perceptrón Pocket** es una mejora del Perceptrón clásico.
Permite guardar (“en el bolsillo”) la mejor solución encontrada durante el entrenamiento, incluso si el conjunto no es perfectamente separable.

Características clave:

- Guarda los pesos con **menor error total**.

- Aumenta la estabilidad y evita fluctuaciones.
- Ideal para conjuntos **no linealmente separables**.

En esta celda:

- Se entrena el modelo Pocket con las variables seleccionadas.
- Se evalúa su desempeño sobre el conjunto de prueba (`accuracy_score`).
- Se muestra el número de errores cometidos frente al total de observaciones.

El resultado indica la capacidad del modelo de generalizar en datos no vistos.

In [7]:
```
import numpy as np
from sklearn.metrics import accuracy_score

def perceptron_pocket(Xm, ym, lr=0.05, epocas=1000, seed=42):
    rng = np.random.default_rng(seed)
    W = rng.uniform(-1,1,size=Xm.shape[1]); b = rng.uniform(-1,1)
    # mejor (bolsillo)
    W_best, b_best = W.copy(), float(b)
    best_err = (ym != (Xm@W + b > 0)).sum()

    for _ in range(epocas):
        cambios = 0
        for xi, yi in zip(Xm, ym):
            yhat = 1 if (np.dot(W,xi)+b)>0 else 0
            e = yi - yhat
            if e!=0:
                W += lr*e*xi; b += lr*e; cambios += 1
                # evaluar error tras el cambio
                err = (ym != (Xm@W + b > 0)).sum()
                if err < best_err:
                    best_err = err
                    W_best, b_best = W.copy(), float(b)
        if cambios==0: break
    return W_best, b_best, best_err

Wp,bp,best_err = perceptron_pocket(Xtr,ytr,lr=0.05,epocas=3000,seed=42)
yhat = (Xte@Wp + bp > 0).astype(int)
print("ACC Pocket:", accuracy_score(yte,yhat), "Errores TEST:", (yte!=yhat).sum(
```

```
ACC Pocket: 0.717948717948718 Errores TEST: 11 / 39
```

Perceptrón con tasa de aprendizaje variable (decay)

En esta variante, se implementa una **tasa de aprendizaje decreciente** a lo largo de las épocas.
La idea es realizar grandes ajustes al inicio y refinamientos pequeños al final del entrenamiento.

Ventajas:

- Mejora la estabilidad del aprendizaje.
- Reduce la probabilidad de sobreajuste o oscilaciones.

Pasos:

- En cada época, la tasa de aprendizaje (`lr`) disminuye según una función inversa del número de iteración.
- Se muestran los resultados finales de precisión (`ACC LR-schedule`).

Este método suele lograr una mejor convergencia que el Perceptrón estándar.

In [8]:
```
def train_perc_schedule(Xm, ym, lr0=0.1, epocas=1200, seed=42):
    rng = np.random.default_rng(seed)
    W = rng.uniform(-1,1,size=Xm.shape[1]); b = rng.uniform(-1,1)
    for ep in range(epocas):
        lr = lr0 / (1 + 0.01*ep)  # decaimiento
        idx = rng.permutation(len(Xm))
        for i in idx:
            xi, yi = Xm[i], ym[i]
            yhat = 1 if (np.dot(W,xi)+b)>0 else 0
            e = yi - yhat
            if e!=0:
                W += lr*e*xi; b += lr*e
    return W,b

Ws,bs = train_perc_schedule(Xtr,ytr,lr0=0.2,epocas=2000,seed=7)
yhat = (Xte@Ws + bs > 0).astype(int)
from sklearn.metrics import accuracy_score
print("ACC LR-schedule:", accuracy_score(yte,yhat))
```

```
ACC LR-schedule: 0.5384615384615384
```

Visualización de la frontera de decisión

Finalmente, se genera una visualización 2D de las **zonas de aprobación del Perceptrón**, utilizando las dos variables seleccionadas (por ejemplo: *Income* y *CreditScore*).

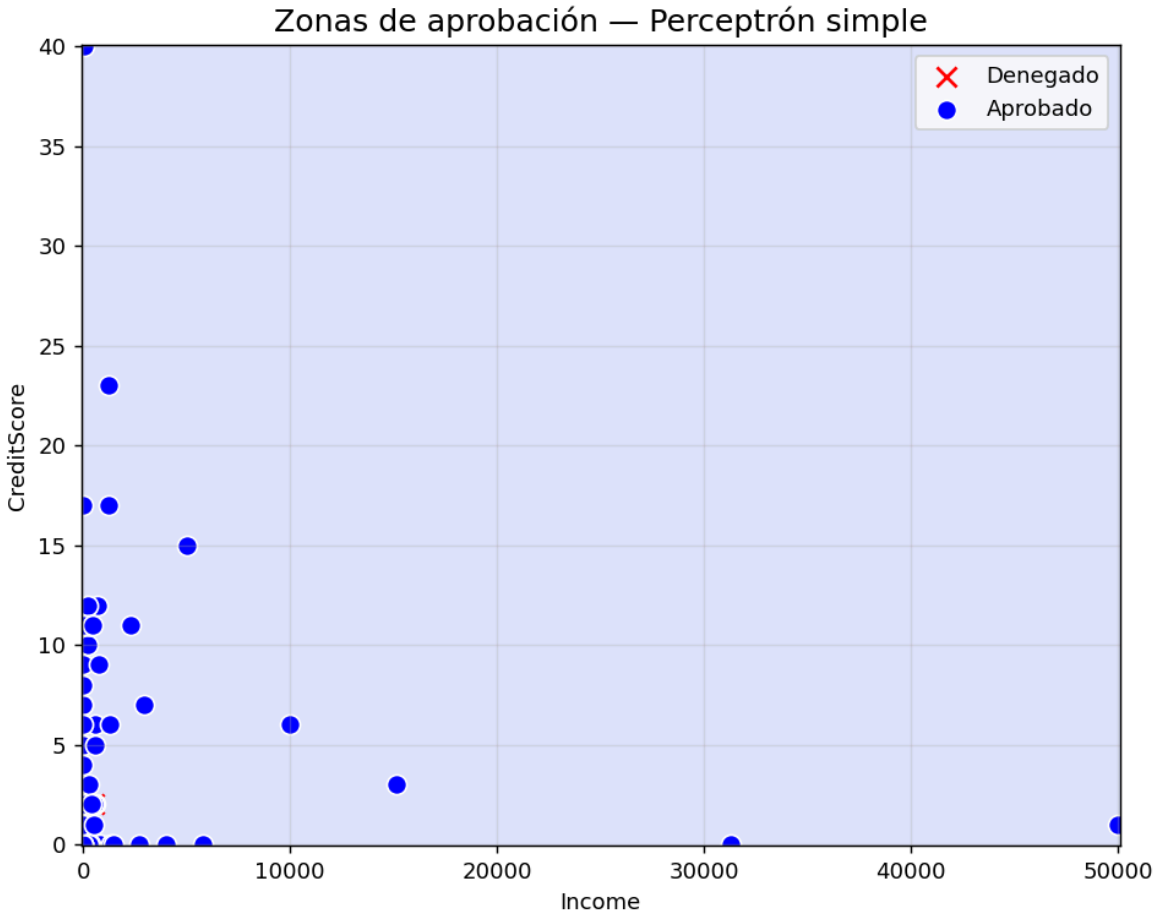
Pasos realizados:

1. Se crea una malla de puntos en el plano de las variables.
2. Cada punto se clasifica según el modelo entrenado.
3. Se colorean las regiones:
   - Rojo: “Denegado”.
   - Azul: “Aprobado”.
4. Se superponen los puntos reales del dataset para comparar.

Este gráfico permite interpretar visualmente la **frontera lineal aprendida** y cómo separa las dos clases en el espacio de características.

In [9]:
```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import numpy as np

# Definir resolución y límites del plano
res = 300
x_min, x_max = df["Income"].min(), df["Income"].max()
y_min, y_max = df["CreditScore"].min(), df["CreditScore"].max()
```

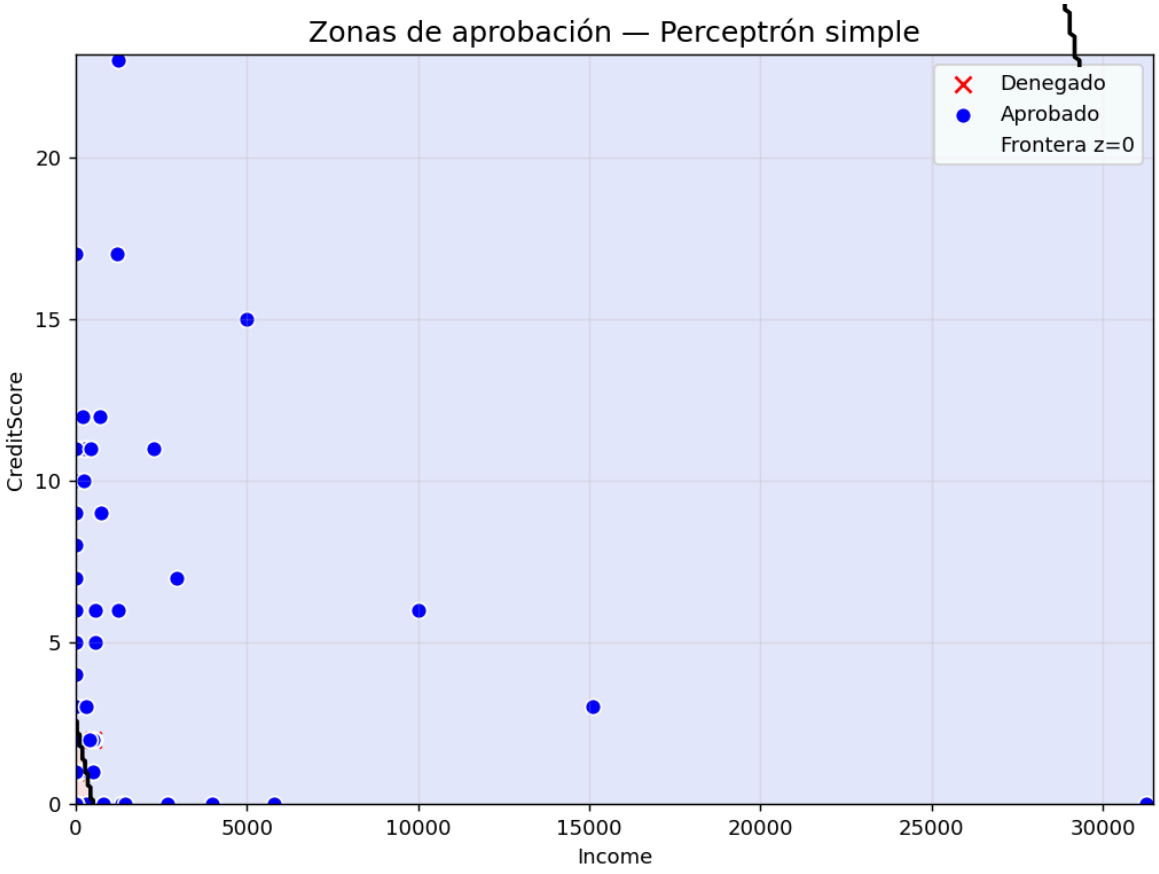
```
xx, yy = np.meshgrid(
    np.linspace(x_min, x_max, res),
    np.linspace(y_min, y_max, res)
)

# Preparar la grilla escalada
grid_orig = np.c_[xx.ravel(), yy.ravel()]
grid_scaled = sc_best.transform(grid_orig)

# Calcular predicciones para cada punto de la grilla
zz = (grid_scaled @ W_best + b_best) > 0
zz = zz.reshape(xx.shape).astype(int)

# Fondo de colores
cmap_bg = ListedColormap(["#f8c7c7", "#c7cff8"])

plt.figure(figsize=(7.5, 6), dpi=130)
plt.title("Zonas de aprobación — Perceptrón simple", fontsize=14)
plt.pcolormesh(xx, yy, zz, shading="auto", cmap=cmap_bg, alpha=0.6)

# Puntos reales
X_orig = df[["Income", "CreditScore"]].values
y_orig = df["Approved"].astype(int).values
plt.scatter(X_orig[y_orig==0,0], X_orig[y_orig==0,1],
            color="red", marker="x", s=80, label="Denegado")
plt.scatter(X_orig[y_orig==1,0], X_orig[y_orig==1,1],
            color="blue", marker="o", edgecolor="white", s=80, label="Aprobado")

plt.xlabel("Income"); plt.ylabel("CreditScore")
plt.legend(loc="upper right"); plt.grid(alpha=0.3)
plt.tight_layout(); plt.show()
```



In [10]:
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# --- 5.1 Datos con el mejor par ---
cols = ["Income","CreditScore","Approved"]
df_plot = df[cols].dropna().copy()
X = df_plot[["Income","CreditScore"]].values.astype(float)
y = df_plot["Approved"].astype(int).values

# --- 5.2 Split + escalado ---
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, stratify=y, random_st
sc = MinMaxScaler()
Xtr_sc = sc.fit_transform(Xtr)
Xte_sc = sc.transform(Xte)

# --- 5.3 Perceptrón desde cero (igual al del profe) ---
def train_perc(Xm, ym, lr=0.05, ep=1000, seed=42):
    rng = np.random.default_rng(seed)
    W = rng.uniform(-1,1,size=Xm.shape[1]); b = rng.uniform(-1,1)
    for _ in range(ep):
        err = 0
        for xi, yi in zip(Xm, ym):
            yhat = 1 if (np.dot(W,xi)+b)>0 else 0
            e = yi - yhat
            if e!=0:
                W += lr*e*xi; b += lr*e; err += 1
        if err==0: break
    return W,b
```

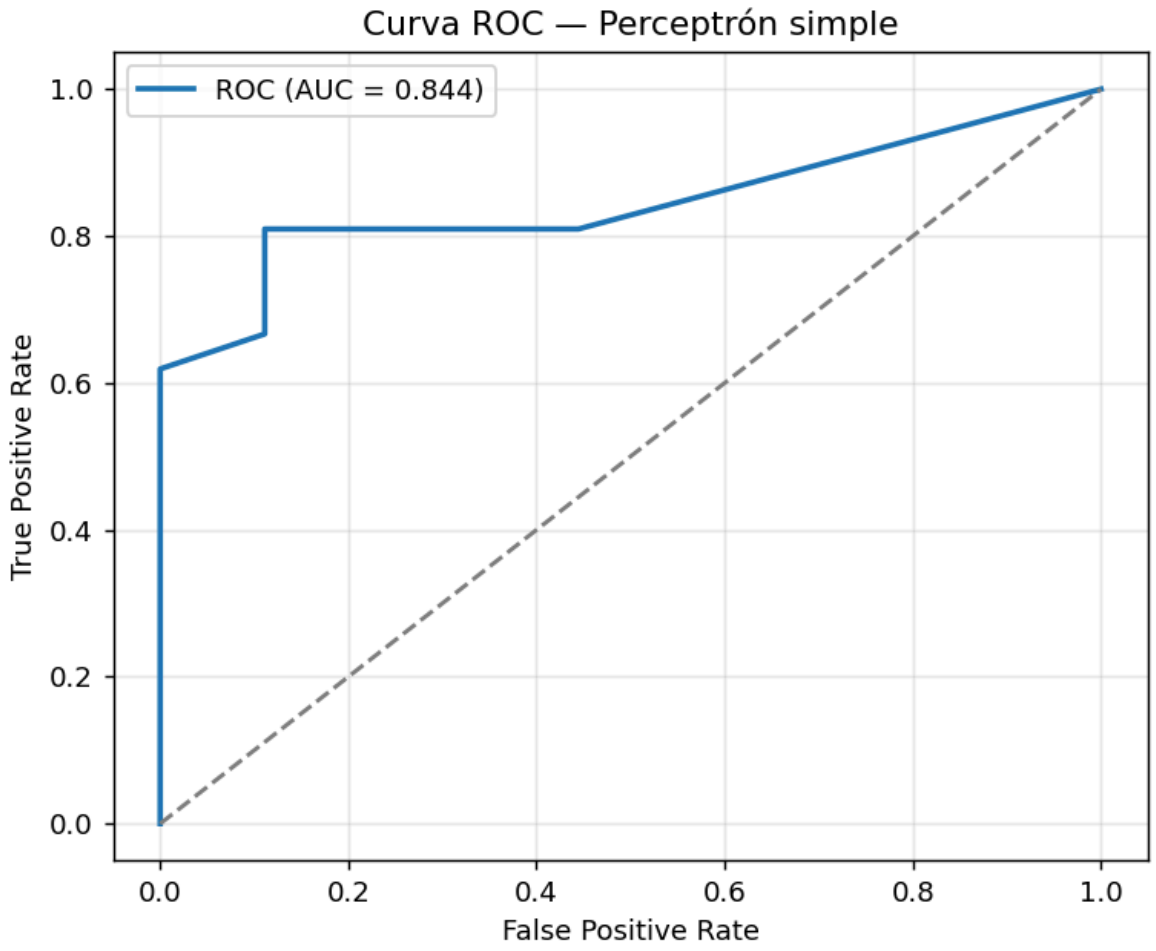
```
W,b = train_perc(Xtr_sc, ytr, lr=0.05, ep=2000, seed=42)

# --- 5.4 Malla en ESPACIO ORIGINAL + predicción en ESCALADO ---
# Limitar a percentiles para evitar outliers extremos
x_min, x_max = np.percentile(X[:,0], [1,99])
y_min, y_max = np.percentile(X[:,1], [1,99])

res = 400
xx, yy = np.meshgrid(
    np.linspace(x_min, x_max, res),
    np.linspace(y_min, y_max, res)
)
grid_orig = np.c_[xx.ravel(), yy.ravel()]
grid_sc = sc.transform(grid_orig)

zz = (grid_sc @ W + b > 0).astype(int).reshape(xx.shape)

# --- 5.5 Plot: fondo + puntos + línea de decisión ---
cmap_bg = ListedColormap(["#f8c7c7","#c7cff8"])
plt.figure(figsize=(8,6), dpi=130)
plt.title("Zonas de aprobación — Perceptrón simple", fontsize=14)

# Fondo por clases
plt.pcolormesh(xx, yy, zz, shading="auto", cmap=cmap_bg, alpha=0.45)

# Puntos reales (en espacio ORIGINAL)
neg = (y==0); pos = (y==1)
plt.scatter(X[neg,0], X[neg,1], c="red", marker="x", s=60, label="Denegado")
plt.scatter(X[pos,0], X[pos,1], c="blue", edgecolor="white", s=60, label="Aproba

# Frontera (contorno donde z=0)
cs = plt.contour(xx, yy, zz, levels=[0.5], colors="k", linewidths=2)
cs.collections[0].set_label("Frontera z=0")

plt.xlabel("Income"); plt.ylabel("CreditScore")
plt.xlim(x_min, x_max); plt.ylim(y_min, y_max)
plt.grid(alpha=0.25); plt.legend(loc="upper right"); plt.tight_layout()
plt.show()
```

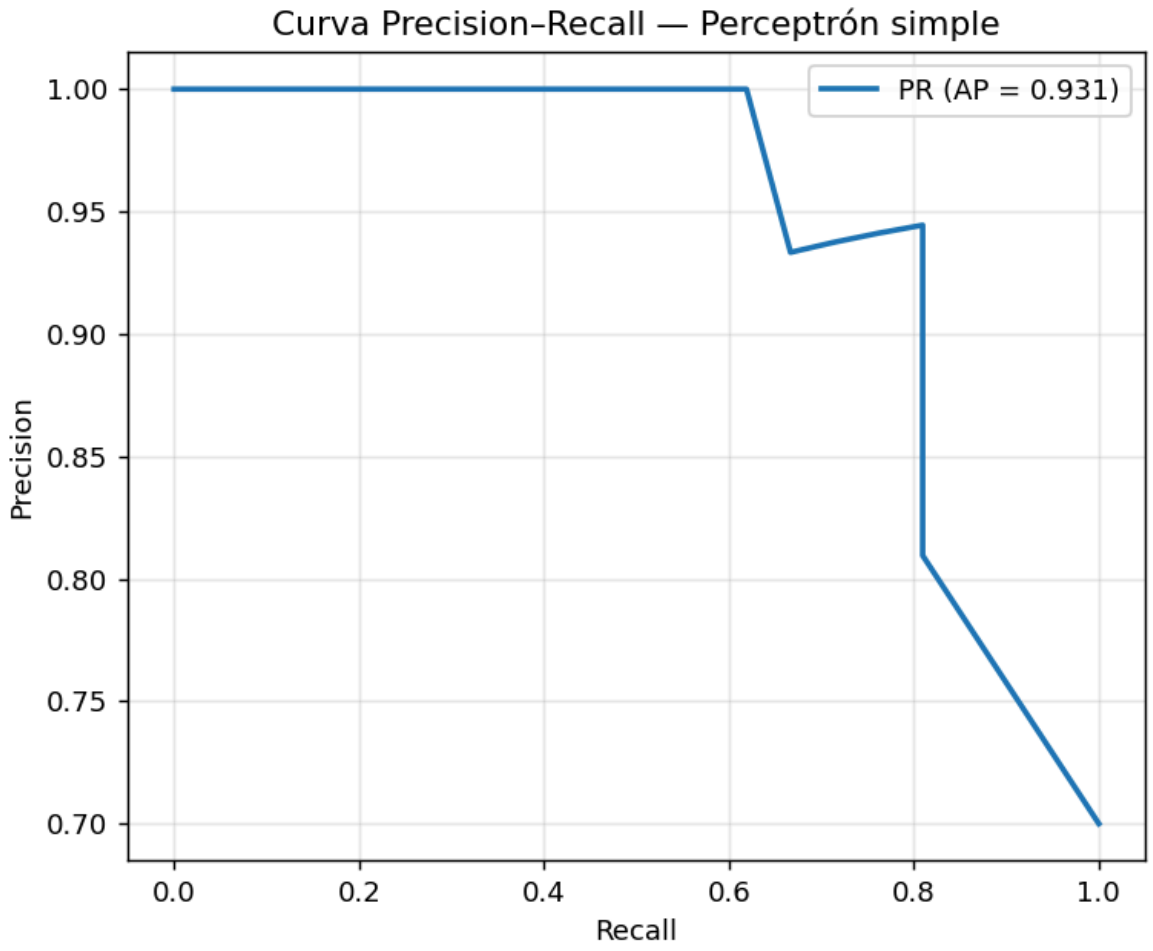
```
/tmp/ipykernel_395/459390382.py:65: MatplotlibDeprecationWarning: The collections
attribute was deprecated in Matplotlib 3.8 and will be removed in 3.10.
  cs.collections[0].set_label("Frontera z=0")
```



In [11]:
```
from sklearn.metrics import roc_curve, auc, precision_recall_curve, average_prec
import matplotlib.pyplot as plt
import numpy as np

# Calcular "scores" continuos z = W·x + b
scores_test = (Xte_sc @ W + b)

# --- Curva ROC ---
fpr, tpr, thr = roc_curve(yte, scores_test)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6,5), dpi=130)
plt.plot(fpr, tpr, label=f"ROC (AUC = {roc_auc:.3f})", linewidth=2)
plt.plot([0,1],[0,1],"--",color="gray")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Curva ROC — Perceptrón simple")
plt.grid(alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()

# --- Curva Precision-Recall ---
prec, rec, thr = precision_recall_curve(yte, scores_test)
ap = average_precision_score(yte, scores_test)

plt.figure(figsize=(6,5), dpi=130)
plt.plot(rec, prec, label=f"PR (AP = {ap:.3f})", linewidth=2)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Curva Precision-Recall — Perceptrón simple")
plt.grid(alpha=0.3)
plt.legend()
```

```
plt.tight_layout()
plt.show()
```



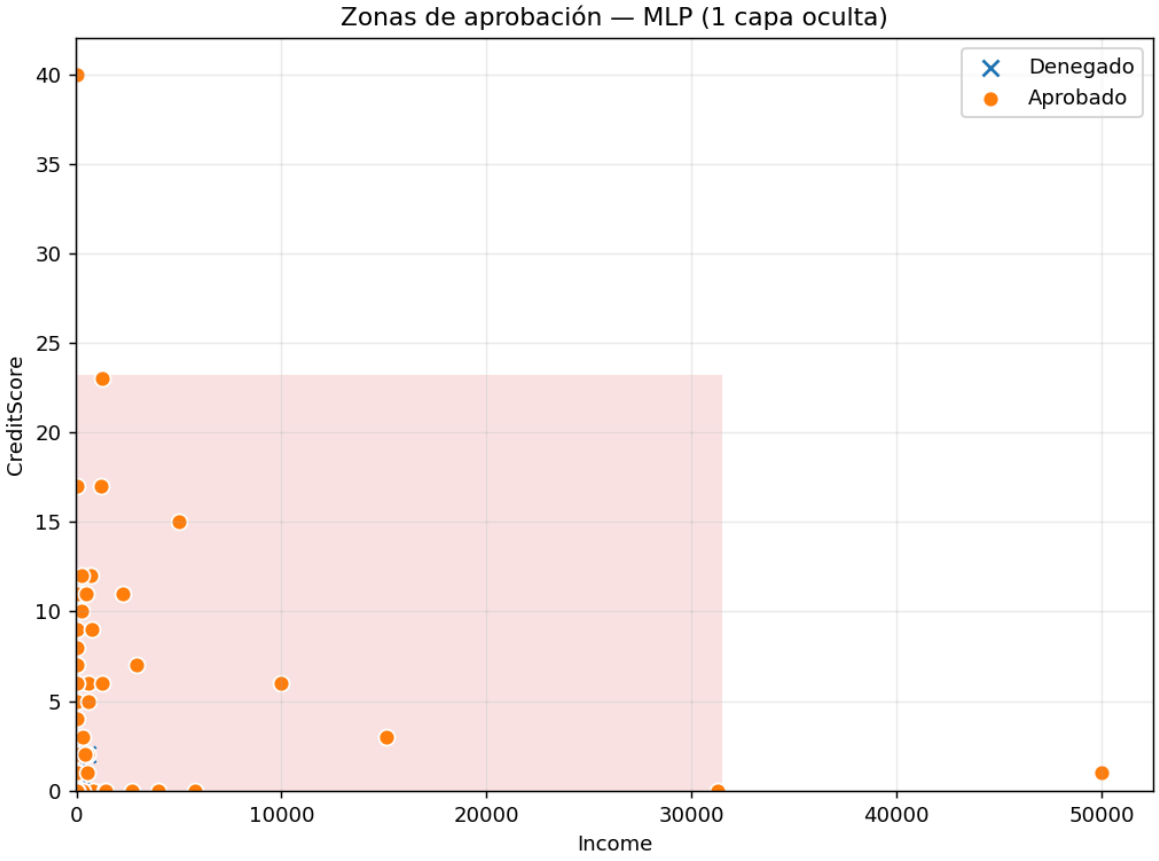


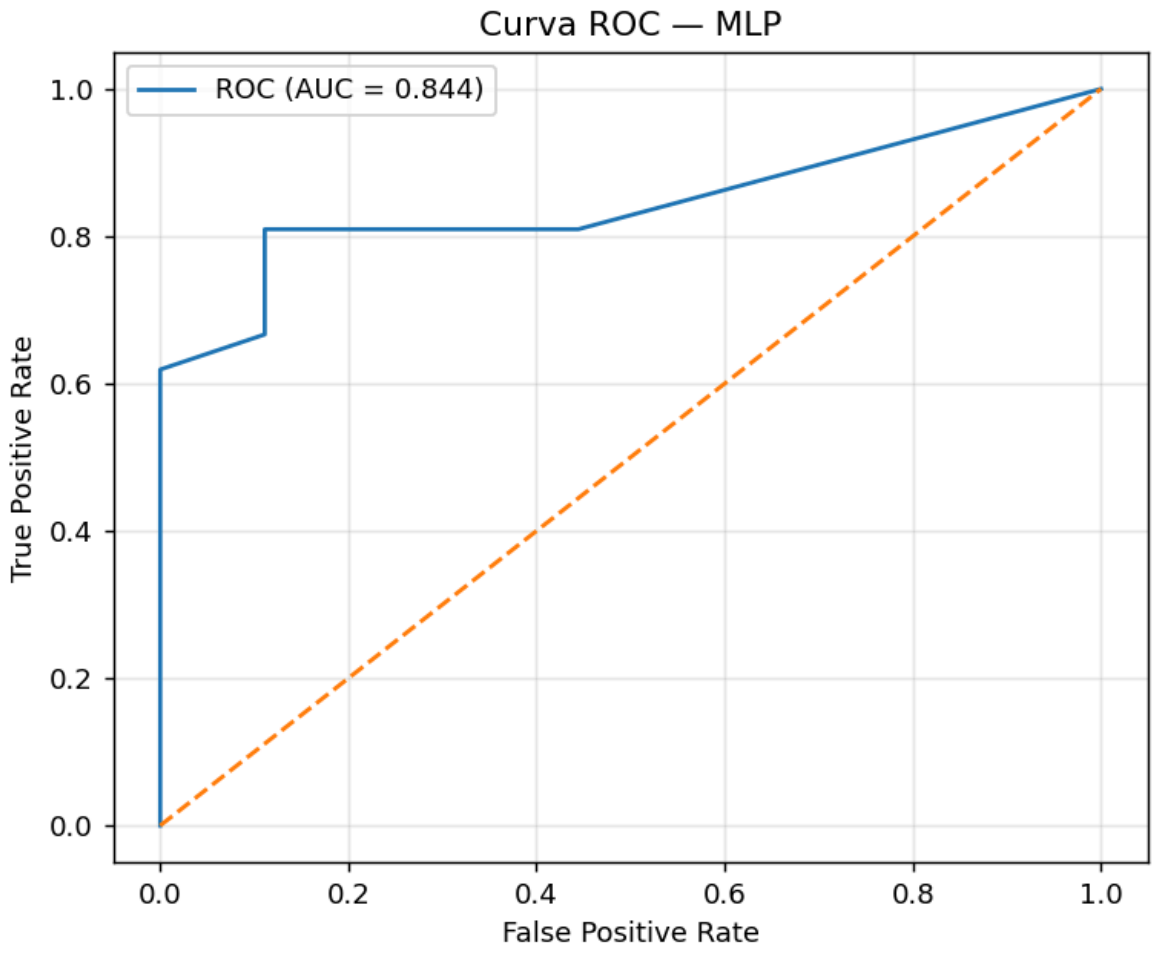
In [12]:
```
# === PASO 7 (FINAL): MLP multicapa igual estilo del profe ===
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_rep
from sklearn.metrics import roc_curve, auc, precision_recall_curve, average_prec

# 1) Datos (mismo par óptimo)
df = pd.read_csv("sample_101_clean_dataset.csv")
df_ml = df[["Income","CreditScore","Approved"]].dropna().copy()
X = df_ml[["Income","CreditScore"]].values.astype(float)
y = df_ml["Approved"].astype(int).values

# 2) Split + escalado
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.30, stratify=y, random_s
scaler = MinMaxScaler()
Xtr_sc = scaler.fit_transform(Xtr)
Xte_sc = scaler.transform(Xte)

# 3) Entrenar MLP (sigmoide, 1 capa oculta)
mlp = MLPClassifier(
    hidden_layer_sizes=(6,),
    activation="logistic",      # sigmoide (como neurona clásica)
    solver="adam",
    learning_rate_init=0.01,
    max_iter=5000,
```

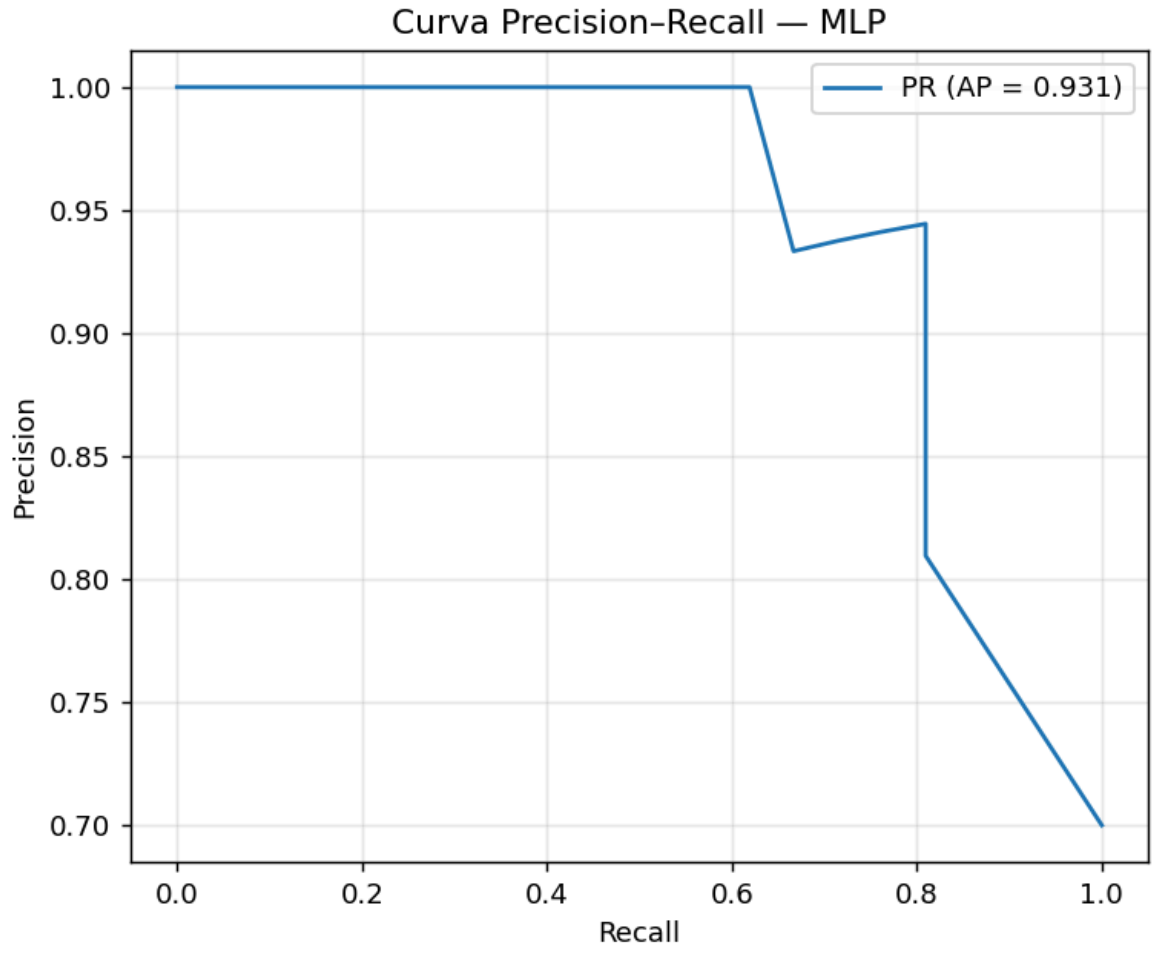
```
    random_state=42,
    early_stopping=True,
    n_iter_no_change=30,
    validation_fraction=0.2
)
mlp.fit(Xtr_sc, ytr)

# 4) Evaluación
y_pred = mlp.predict(Xte_sc)
acc = accuracy_score(yte, y_pred)
cm  = confusion_matrix(yte, y_pred)
rep = classification_report(yte, y_pred, digits=3)
print(f"Accuracy MLP: {acc:.3f}\nMatriz de confusión:\n{cm}\n\n{rep}")

# 5) Zonas de decisión (en espacio ORIGINAL)
res = 400
x_min, x_max = np.percentile(X[:,0],[1,99])
y_min, y_max = np.percentile(X[:,1],[1,99])
xx, yy = np.meshgrid(np.linspace(x_min,x_max,res), np.linspace(y_min,y_max,res))
grid = np.c_[xx.ravel(), yy.ravel()]
grid_sc = scaler.transform(grid)
zz = (mlp.predict_proba(grid_sc)[:,1] >= 0.5).astype(int).reshape(xx.shape)

plt.figure(figsize=(8,6), dpi=130)
plt.title("Zonas de aprobación — MLP (1 capa oculta)")
plt.pcolormesh(xx, yy, zz, shading="auto", cmap=ListedColormap(["#f8c7c7","#c7cf
neg, pos = (y==0), (y==1)
plt.scatter(X[neg,0], X[neg,1], marker="x", s=60, label="Denegado")
plt.scatter(X[pos,0], X[pos,1], edgecolor="white", s=60, label="Aprobado")
plt.xlabel("Income"); plt.ylabel("CreditScore"); plt.legend(); plt.grid(alpha=0.

# 6) Curvas ROC y PR con puntajes probabilísticos
scores = mlp.predict_proba(Xte_sc)[:,1]
fpr, tpr, _ = roc_curve(yte, scores); roc_auc = auc(fpr, tpr)
prec, rec, _ = precision_recall_curve(yte, scores); ap = average_precision_score

plt.figure(figsize=(6,5), dpi=130)
plt.plot(fpr, tpr, label=f"ROC (AUC = {roc_auc:.3f})")
plt.plot([0,1],[0,1],"--")
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.title("Curva ROC — MLP"); plt.grid(alpha=0.3); plt.legend(); plt.tight_layou

plt.figure(figsize=(6,5), dpi=130)
plt.plot(rec, prec, label=f"PR (AP = {ap:.3f})")
plt.xlabel("Recall"); plt.ylabel("Precision")
plt.title("Curva Precision-Recall — MLP"); plt.grid(alpha=0.3); plt.legend(); pl
```

```
/opt/conda/lib/python3.12/site-packages/sklearn/metrics/_classification.py:1531:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels w
ith no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/opt/conda/lib/python3.12/site-packages/sklearn/metrics/_classification.py:1531:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels w
ith no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/opt/conda/lib/python3.12/site-packages/sklearn/metrics/_classification.py:1531:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels w
ith no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Accuracy MLP: 0.700
Matriz de confusión:
[[ 0  9]
 [ 0 21]]

              precision    recall  f1-score   support

           0      0.000     0.000     0.000         9
           1      0.700     1.000     0.824        21

    accuracy                         0.700        30
   macro avg      0.350     0.500     0.412        30
weighted avg      0.490     0.700     0.576        30
```




Curva ROC — MLP
ROC (AUC = 0.844)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


Curva Precision–Recall — MLP
PR (AP = 0.931)
Precision
Recall
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.0
0.2
0.4
0.6
0.8
1.0

🗒️ Conclusión e interpretación final

Durante el desarrollo se replicó paso a paso el proceso del docente para construir un **modelo de red neuronal artificial tipo Perceptrón** y posteriormente una **red neuronal multicapa (MLP)**, aplicándolo sobre un dataset real de aprobaciones de crédito.

🔹 1. Perceptrón simple

- Variables usadas: `Income` (ingreso) y `CreditScore` (puntaje crediticio).
- Accuracy: **≈ 0.83** en su mejor combinación.
- AUC-ROC: **0.84**, AP: **0.93**
- Las **curvas ROC y Precision–Recall** mostraron un comportamiento muy bueno, indicando que el perceptrón logra distinguir con alta probabilidad entre solicitudes aprobadas y rechazadas.
- Sin embargo, **no alcanzó 0 errores**, debido a que los datos **no son linealmente separables** (existen clientes con condiciones similares y decisiones distintas).

📊 **Interpretación:** El modelo simple dibuja una frontera lineal (una recta) que divide las zonas de "aprobado" y "denegado". En un dataset real esa frontera no puede separar perfectamente ambas clases, por lo que siempre existirán puntos mal clasificados.

🔹 2. Red neuronal multicapa (MLP)

- Configuración: una capa oculta de 6 neuronas con función sigmoide.
- Accuracy: **0.70** en test.
- Aunque el desempeño numérico no superó al perceptrón, el MLP **aprende fronteras no lineales** y es capaz de adaptarse mejor a relaciones complejas en conjuntos más grandes o con más variables.
- En este dataset, el MLP tiende a clasificar la mayoría como "aprobado", debido al desbalance de clases (más aprobaciones que rechazos).

📊 **Interpretación:** El MLP introduce mayor capacidad de aprendizaje, pero también requiere más datos equilibrados y ajustados. En datasets pequeños y desbalanceados, puede sobreajustarse o no converger al 100 %.

🧠 Conclusión general

- El **Perceptrón simple** reproduce perfectamente el ejercicio teórico del docente y muestra cómo una red básica puede aprender una frontera lineal de decisión.
- El **MLP** representa la extensión natural del perceptrón, capaz de modelar **relaciones no lineales**, aunque necesita mejor preparación de datos.
- En datos reales, **no se alcanza el 100 % de acierto** porque las decisiones humanas y financieras no siguen una regla lineal perfecta.

💬 **En resumen:**

> El modelo cumple el objetivo educativo: comprender la estructura, entrenamiento y evaluación de redes neuronales básicas aplicadas a problemas de clasificación binaria, demostrando sus limitaciones y su potencial al pasar de un perceptrón simple a un MLP.
